Time–Memory–Processor Trade-Offs

HAMID R. AMIRAZIZI AND MARTIN E. HELLMAN, FELLOW, IEEE

Abstract—**It is demonstrated that usual time–memory trade-offs offer no asymptotic advantage over exhaustive search. Instead, trade-offs between time, memory, and parallel processing are proposed. Using this approach it is shown that most searching problems allow a trade-off between C_s, the cost per solution, and C_m, the cost of the machine: doubling C_m increases the solution rate by a factor of four, halving C_s. The machine which achieves this has an unusual architecture, with a number of processors sharing a large memory through a sorting/switching network. The implications for cryptanalysis, the knapsack problem, and multiple encryption are discussed.**

I. Introduction

MANY SEARCHING tasks, such as the knapsack [1] and discrete logarithm problems [2], allow time–memory trade-offs. That is, if there are N possible solutions to search over, the time–memory trade-off allows the solution to be found in T operations (time) with M words of memory, provided the time–memory product TM equals N. (Often the product is of the form $cN \log_2 N$, but for simplicity we neglect logarithmic and constant factors.)

The practical value of a time–memory trade-off is apparent if one considers solving a discrete logarithm problem with 2^{50} possible solutions. Exhaustive search would require 2^{50} operations and would take almost 36 years of CPU time, even at 1 μs per trial. Using the time–memory trade-off on such a CPU with $T = 2^{30}$ and $M = 2^{20}$ is much more cost-effective, requiring only 18 min of CPU time on a machine with 1 million words of memory, typical of current minicomputers.

While in this example the cost savings is real, extending the approach to truly large problems does not produce ever increasing savings. Asymptotically, the cost of running a time–memory trade-off program grows linearly in N for the following reason. As N becomes ever larger, eventually the cost of the memory (growing as some power of N) dominates the cost of the single CPU so that the capital cost of the machine C_m grows linearly in M. C_r, the cost of running the program, is proportional to the product of C_m and the run time T of the program, and therefore grows as MT. Therefore, $C_r \sim MT = N$, the same cost as for exhaustive search. ($X \sim Y$ means X is proportional to Y, neglecting constant and logarithmic factors.) Exhaustive search can be run on a machine of fixed size, so, neglecting constant factors, C_m can be taken as unity and $C_r \sim C_m T \sim 1 \cdot N = N$.

To formalize, assume that c_p is the cost of one processor, c_m is the cost of one word of memory (or bit, since constant and logarithmic factors are neglected), and P and M are the number of processors and words of memory used by the machine, so that

$$C_m = c_p P + c_m M. \tag{1}$$

However, c_p and c_m are just constant factors, which are being neglected, so (1) can be simplified to

$$C_m \sim \max(P, M). \tag{2}$$

Equations (1) and (2) assume that memory and processors are the dominant cost of the machine. Later, when additional components are added (e.g., a switching network so any processor can access any word of memory), it must be ensured that their costs do not dominate the cost of the machine by more than the constant or logarithmic factors which are being neglected.

Using (2), C_r, the cost per run, is

$$C_r \sim C_m T \sim \max(P, M) \cdot T \tag{3}$$

and C_s, the cost per solution, is

$$C_s = C_r / S \sim \max(P, M) \cdot T/S \tag{4}$$

where S is the simultaneity of solution, the number of problems solved simultaneously in one run. It is instructive to plot the (C_m, C_s) points for a time–memory trade-off. This is shown in Fig. 1.

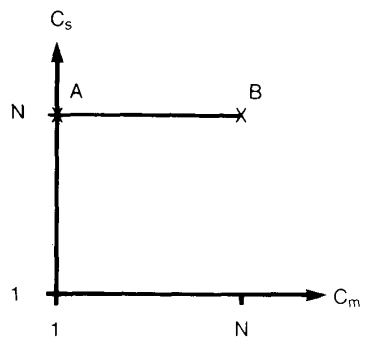

Fig. 1. $(C_m - C_s)$ curve for usual TM trade-off.

Manuscript received December 12, 1983; revised March 3, 1987. This work was supported in part by the National Security Agency under Contract MDA904-81-C-0414, and by the National Science Foundation under Grants ENG 10173 and ECS-8307741. This paper was presented in part at the Crypto '81 Conference, Santa Barbara, CA, August 1981, and at the IEEE International Symposium on Information Theory, Les Arcs, France, June 1982.

H. R. Amirazizi is with NEC America, Inc., San Jose, CA 95134.

M. E. Hellman is with the Department of Electrical Engineering, Stanford University, Stanford, CA 94305.

IEEE Log Number 8821590.

0018-9448/88/0500-0505$01.00 ©1988 IEEE

1) Exhaustive search has $P = M = 1$, $T \sim N$, $S = 1$, so $C_m \sim 1$ and $C_s \sim N$ as indicated in point A.

2) The usual form of table lookup has $P = 1$, $M = N$, and (neglecting precomputation, and constant and logarithmic factors) $T = 1$. With $S = 1$, $C_m \sim N$ and $C_s \sim N$ as indicated in point B. (While this brief analysis neglected precomputation, later in this paper precomputation effort will be included.)

3) Usual time–memory trade-offs have $P = 1$, so $C_m \sim \max(P, M) = M$. Because $S = 1$ and $MT = N$, it follows that $C_s \sim N$. As M varies from 1 to N, the usual time–memory trade-off traces out the horizontal line connecting points A and B.

Under this simplified or asymptotic model, time–memory trade-offs have no advantage over exhaustive search or table lookup. They have the same cost per solution as exhaustive search because their increased speed is offset by their use of more expensive machines. Exhaustive search can do the same by recourse to parallelism. Using P parallel processors increases C_m and decreases T each by a factor of P so that C_s is unchanged.

Later, when we discuss different trade-offs we will always plot C_s versus C_m. However, these trade-offs also are affected by a third variable, simultaneity, which is proportional to time delay to obtain a solution. To make the point clear, consider two algorithms that have the same (C_m, C_s) points, but different simultaneities. In algorithm a) let us assume as an example that $C_m \sim N^{1/2}$, $T \sim 1$, $S = 1$, and therefore $C_s \sim N^{1/2}$. In algorithm b) assume that $C_m \sim N^{1/2}$, $T \sim N^{1/2}$, and $S = N^{1/2}$, also resulting in $C_s \sim N^{1/2}$. Clearly, a) can imitate the performance of b) by just solving $N^{1/2}$ problems in $N^{1/2}$ units of time and outputting the solutions as a batch. However, b) cannot perform as a) because it has a longer time delay. Therefore, while both a) and b) are represented by the same (C_m, C_s) points, a) has a lower simultaneity and time delay; thus a) is a better algorithm.

II. Table Lookup Time–Memory–Processor Trade-Off

Time–memory trade-offs have usually been regarded as much more cost-effective than exhaustive search, and our earlier discrete logarithm problem with $N = 2^{50}$ certainly benefited from using that approach. How is this to be reconciled with the foregoing reasoning which shows that, asymptotically, time–memory trade-offs are no more cost-effective than exhaustive search?

The answer lies in the fact that because the asymptotic analysis neglected constant factors, it neglected the difference between c_p and c_m, the costs of a single processor and of a single word of memory. On today's minicomputers c_p is approximately \$2000 while c_m is on the order of \$0.002. This millionfold difference allows a cost saving of approximately one million by use of a time–memory trade-off, but asymptotically that is "only" a constant factor.

To obtain better performance, we introduce the concept of a time–memory–processor trade-off and show that for any searching problem a cost per solution of one ($C_s \sim 1$) can be achieved by using table lookup with N processors instead of the usual single processor. This does not change $C_m \sim \max(P, M)$ because M already equals N for table lookup. The N processors, $P_1, P_2, \cdots, P_N$ are used to work on N separate problems simultaneously and share the N words of memory, $M_1, M_2, \cdots, M_N$ through a "sorting/switching network" as indicated in Fig. 2.

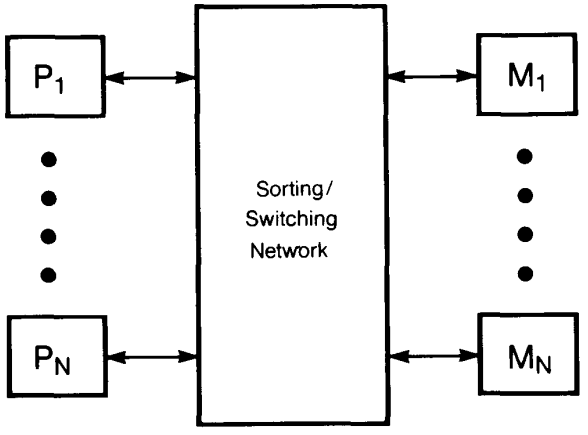

Fig. 2. Cost-effective table lookup machine architecture.

The sorting/switching network can connect any one of the N processors to any one of the N words of memory and vice versa. This can be done either in switching mode (e.g., processor i can ask for the contents of memory word j or ask that a computed value be stored in memory word j) or in sorting mode (e.g., if each processor computes a value Y_i, the network can sort these N values and store the smallest in memory word 1 and the largest in memory word N).

With this architecture, table lookup can be accomplished in "unit" time (as usual, neglecting constant and logarithmic factors) even if the precomputation time is included. For simplicity, assume that the function $Y = f(X)$ to be inverted is a one-to-one correspondence between the set $\{1, 2, 3, \cdots, N\}$ and itself, as in the discrete logarithm problem when the base of the logarithm is a primitive element. In Section III we will extend our results to inverting a function f with arbitrary range.

Processor i computes $Y_i = f(i)$ in unit time (assuming $f(X)$ can be computed in constant time, and neglecting constant factors). Then the network is used in switching mode to direct the pair (i, Y_i) to memory location Y_i where i is stored. Batcher [7] has shown that the time to compute the required state of the switching network grows only as a logarithmic factor in $N(O(\log^2 N))$ and hence is "negligible." He also shows that the cost of the network measured in terms of switching elements is $O(N \log^2 N)$, and, therefore, it does not dominate the cost of the machine by more than constant or logarithmic factors which are being ne-

glected. (There is some question if the cost includes wire length [9].) Batcher's switching network is "self-organizing," i.e., the computing within the network is distributed and it does not require any centralized control. Other switching networks exist [8] that have smaller number of switching elements ($O(N \log N)$), but they require central control for setting up the connections.

Then N problems are presented to the N processors, with the problem presented to the ith processor being of the form: "find $X^{(i)}$, an inverse image of the value $Y^{(i)}$." The network is again used in switching mode to connect processor i to memory location $Y^{(i)}$ which contains the answer.

When storing the results of the precomputation, the switching network connects exactly one processor to each word of memory because of our assumption that $f(\cdot)$ is a one-to-one correspondence. However, during the answer-finding phase this may not be the case. Several or even all of the processors may have been presented with the same problem and need to access the same word of memory. The switching network must therefore be "multiaccess," allowing a memory location to be accessed by as many processors as need its contents. Such a network can be constructed using Batcher's sorting network as a base as described in [9]. As with Batcher's network, the multi-access network is "self-organizing," has the same amount of time delay ($O(\log^2 N)$), and has the same number of switching elements ($O(N \log^2 N)$).

Modifying table lookup in the described manner produces a time–memory–processor trade-off with $C_m \sim N$, $T \sim 1$ (including precomputation), and $S = N$ so that $C_s \sim C_m T/S \sim 1$. While at first this architecture may appear unreasonable because it has a simultaneity of N, we now show how to obtain more reasonable time–memory–processor (TMP) trade-offs based on this architecture.

III. General Searching Problem TMP Trade-Off

This section develops an algorithm for solving the following problem: given K values, $Y^{(1)}, \cdots, Y^{(K)}$, find the corresponding $X^{(1)}, \cdots, X^{(K)}$ such that $Y^{(K)} = f(X^{(K)})$. We allow f to be an arbitrary mapping from the set $\{1, \cdots, N\}$ to a set of arbitrary size. For example, in cryptography f may be a mapping from N keys to one of L messages. When constant and logarithmic factors are neglected our algorithm allows a TMP trade-off with the following parameters: $C_m \sim K$, $S = K$, $T \sim N/K$, and $C_s \sim N/K$ for $1 \le K \le N$.

The architecture of the machine which achieves this performance has K processors connected to K words of memory through a sorting/switching network. Each processor has several words of local memory for storing $Y^{(i)}$, $X^{(i)}$ (when found), i, K, t, etc. The following steps summarize the algorithm:

1) Initialization: processor i is told $Y^{(i)}$ and sets $t = 0$. (K and i are wired in.)
2) The ith processor computes $Y_i = f(i + Kt)$ as part of a "precomputation."
3) The pairs $\langle Y_i, i + Kt \rangle$ are presented to the sorting/switching network which sorts them on the first field and stores the sorted values in memory locations 1 through K. This allows a table lookup for any of the K values $X^{(i)}$ such that $Kt < X^{(i)} \le K(t+1)$.
4) Each processor performs a binary search on the memory to find if a pair of the form $\langle Y^{(i)}, X^{(i)} \rangle$ is stored in memory. Processor i is successful for this value of t if $Kt < X^{(i)} \le K(t+1)$, in which case $X^{(i)}$ is stored in the local memory as the solution.
5) Increment t to $t+1$: if $t = N/K$, stop; else go back to step 2.

There are K processors used in the foregoing algorithm. The number of words of memory that is required is $O(K)$, and even if $f(\cdot)$ is many to one, the sorting/switching network requires $O(K \log^2 K)$ elements [7], [9]. Therefore, neglecting constant and logarithmic factors, C_m, the cost of the machine, is K. Because K problems are solved at the same time, the simultaneity S is also equal to K.

The "precomputation" of step 2) of the algorithm is done in parallel using K processors and, therefore, takes unit time. Steps 3) and 4) use the sorting/switching network and as shown in [7], [9] take a time of $O(\log^2 K)$. Thus neglecting constant and logarithmic factors, each iteration of the algorithm takes only unit time. Because there are at most N/K iterations, the total time to find the solutions is $T \sim N/K$. Hence the cost per solution C_s becomes $C_s \sim C_m T/S \sim (K \cdot N/K)/K = N/K$. Note that $C_m C_s \sim N$, and, therefore, our algorithm allows a trade-off between C_m and C_s. This means that increasing the cost of the machine by a factor c increases the solution rate by c^2, thereby decreasing the cost per solution by a factor of c.

The (C_m, C_s) points and the corresponding values of S are plotted in Fig. 3 on a logarithmic scale. The advantage of this TMP trade-off over a usual TM trade-off is seen by comparing Figs. 1 and 3.

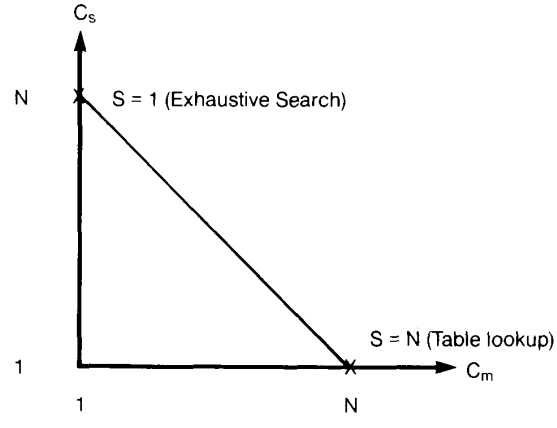

Fig. 3. TMP trade-off for searching (N possible solutions).

IV. Time–Memory–Processor Trade-Off for Binary Knapsacks

This section describes an algorithm for the knapsack problem that has a better performance than the general TMP trade-off of Section III. Although it has the same $(C_m - C_s)$ curve as the general TMP trade-off, it allows lower simultaneity ($S=1$ for half the curve and S increases smoothly from 1 to N for the rest of the curve, whereas S increases smoothly from 1 to N over the entire curve for the general TMP trade-off.)

The binary knapsack problem is as follows. Given an integer Y and an n-tuple $\boldsymbol{a}$ with positive integer components, $\boldsymbol{a} = (a_1, \cdots, a_n)$, find a binary vector $\boldsymbol{X}$ such that $Y = \boldsymbol{a}^* \boldsymbol{X}$ (* denotes the dot product). A simple time–memory trade-off [1] exists for this problem that leads to $T \sim 2^{(1-\alpha)n}$ and $M \sim 2^{\alpha n}$ for $0 \le \alpha \le 1$ and has $P = S = 1$. As shown in Fig. 1, this trade-off has the same cost per solution as exhaustive search ($C_s \sim 2^n = N$) and therefore is of no asymptotic value.

Schroeppel and Shamir [10] found a method that has $P = S = 1$, $T \sim 2^{(1+\alpha_s)n/2}$, $M \sim 2^{(1-\alpha_s)n/4}$, and hence has $C_m \sim \max(P, M) = M = 2^{(1-\alpha_s)n/4}$, resulting in $C_s \sim 2^{(3+\alpha_s)n/4}$, where $0 \le \alpha_s \le 1$. Although their method produces a lower cost per solution than exhaustive search ($\min C_s \sim 2^{3n/4}$ at $\alpha_s = 0$), we have not been able to extend it to smaller values of cost per solution.

In this section, we derive a TMP trade-off for the knapsack problem that corresponds to $C_m \sim 2^{(1-\alpha)n/2}$, $T \sim 2^{\alpha n}$, $S = 1$, and $C_s \sim 2^{(1+\alpha)n/2}$ for $0 \le \alpha \le 1$. By varying the parameter α, this trade-off gives all values of cost per solution from 2^n to $2^{n/2}$ while as usual, $C_m C_s \sim N$. The advantage of this algorithm and machine over the general TMP trade-off of Section III is its low value of simultaneity $S = 1$.

If we wish to continue along the $C_m C_s \sim N$ line beyond $C_m \sim 2^{n/2}$, either T must be less than one (impossible) or S must be greater than one. Therefore, to get a cost per solution of less than $2^{n/2}$, we must have $S > 1$. We extend the TMP trade-off so that its simultaneity increases smoothly from 1 to 2^n as the cost of the machine goes up from $2^{n/2}$ to 2^n. The time remains at one, and the cost per solution decreases from $2^{n/2}$ to 1. While this algorithm also has the same $(C_m - C_s)$ curve as the general TMP trade-off of Section III, it is better because it has lower simultaneity, or equivalently, a shorter time delay.

We first describe the algorithm for $S = 1$ (the case for which $S > 1$ is similar and is given in [9]). The basic idea is to divide the $\boldsymbol{a}$ vector into three parts of lengths: αn, $(1-\alpha)n/2$, and $(1-\alpha)n/2$. Then we generate all $2^{(1-\alpha)n/2}$ possible subset sums from the second part of the $\boldsymbol{a}$ vector and all $2^{(1-\alpha)n/2}$ possible subset sums from the third part of the $\boldsymbol{a}$ vector. If we knew the correct value (subset sum) from the first part (call it $Y1$), then to solve the problem we would need to find a value from the second part (say $Y2_j$) and one from the third part (say $Y3_k$) such that $Y1 + Y2_j + Y3_k = Y$ or $Y2_j = Y - Y1 - Y3_k$. This can be checked quickly using the sorting/switching network that looks for a match between the $\{Y2_j\}$ and $\{Y - Y1 - Y3_k\}$ tables in unit time. Because we do not know the correct $Y1$, we do an exhaustive search on the first part of the $\boldsymbol{a}$ vector. This is given in more detail in the following algorithm. Note that in the algorithm, i and j are integers while $\boldsymbol{i}$ and $\boldsymbol{j}$ are binary vectors obtained from the binary representations of i and j.

1) Initialize:

$\boldsymbol{U} = (a_1, \cdots, a_{\alpha n})$, [first part of length αn]

$\boldsymbol{V} = (a_{\alpha n+1}, \cdots, a_{(1+\alpha)n/2})$, [second part of length $(1-\alpha)n/2$]

$\boldsymbol{W} = (a_{(1+\alpha)n/2+1}, \cdots, a_n)$ [third part of length $(1-\alpha)n/2$]

$i = 0$, [αn-bit binary integer]

$M_1 = 2^{\alpha n}$

$M_2 = 2^{(1-\alpha)n/2}$.

2) Generate the subset sums from the second part in unit time by using $M_2 = 2^{(1-\alpha)n/2}$ processors:

FOR $j_1 = 0$ to $M_2 - 1$, DO

$P_{j_1} = \langle \boldsymbol{V}^* \boldsymbol{j}_1, j_1, \text{"second"} \rangle$.

3) Compute a value from the first part:

IF $i < M_1$, THEN

$Y1 = \boldsymbol{U}^* \boldsymbol{i}$

ELSE "stop".

4) Generate the subset sums from the third part in unit time by using M_2 processors:

FOR $j_2 = 0$ to $M_2 - 1$, DO

$P_{M_2+j_2} = \langle Y - Y1 - \boldsymbol{W}^* \boldsymbol{j}_2, j_2, \text{"third"} \rangle$.

5) Check to see if there is a match:

a) Sort the $2M_2\{P_j\}$ values produced in 2 and 4 on the first field (using the sorting/switching network) and store them in the memory locations 1 through $2M_2$.

b) Check the memory for adjacent items with the same first field, that is, of the form:

$\langle \text{value}, j_1, \text{"second"} \rangle$

$\langle \text{value}, j_2, \text{"third"} \rangle$.

This can be accomplished in unit time by having a comparator connected to each pair of adjacent memory cells. The cost of this additional circuitry only affects C_m by a constant factor and is therefore neglected.

6) Output the solution or iterate the algorithm:

IF (match exists), THEN:

$\boldsymbol{X} = \boldsymbol{i} \# \boldsymbol{j}_1 \# \boldsymbol{j}_2$

(# stands for binary concatenation)

GOTO step 7

IF (no match), THEN:

$i = i + 1$

GOTO step 3.

7) For the case that the knapsack is not one to one, other possible solutions can be found by continuing the algorithm:

$i = i + 1$

GOTO step 3.

It is easy to see that $O(2^{(1-\alpha)n/2})$ processors and words of memory are used in all steps of the algorithm. As shown in [7] the sorting/switching network requires $O(2^{(1-\alpha)n/2} \log^2(2^{(1-\alpha)n/2}))$ elements. Therefore, neglecting constant and logarithmic factors the cost of the machine is $C_m = 2^{(1-\alpha)n/2}$. Since only one problem is solved, the simultaneity S is equal to one.

Steps 2 and 4 of the algorithm are done in parallel and take only unit time. Step 5 does a sorting of the processor values that takes a logarithmic time [7] and a searching of the memory that takes unit time. Thus neglecting constant and logarithmic factors, each iteration of the algorithm takes only unit time. Because there are $2^{\alpha n}$ iterations, the total time T is $2^{\alpha n}$. Therefore, the cost per solution $C_s \sim C_m T/S \sim 2^{(1-\alpha)n/2} 2^{\alpha n}/1 = 2^{(1+\alpha)n/2}$.

We see that $C_m C_s \sim 2^n$, and thus we get the same $(C_m - C_s)$ curve as in the general TMP trade-off of the previous section. However, as mentioned earlier, this is a better trade-off because it has $S = 1$, and therefore, for fixed (C_m, C_s) points it takes less time to find the solution. In Fig. 4 we plot C_s versus C_m on a logarithmic scale for $0 \le \alpha \le 1$.

The dotted portion of the curve corresponds to an algorithm described in [9] for which the simultaneity S increases from 1 to 2^n, the cost of the machine C_m increases from $2^{n/2}$ to 2^n, the time T remains at one, and the cost per solution decreases from $2^{n/2}$ to 1. For example, at point C we have $S = 2^{n/2}$, $C_m \sim 2^{3n/4}$, $T \sim 1$, and $C_s \sim 2^{n/4}$. Note that it too has $C_m C_s \sim 2^n$, and therefore, it is an extension of the curve for $S = 1$.

Remarks: 1) Point A in the Fig. 4 corresponds to the usual time–memory trade-off with $M \sim 2^{n/2}$, $P = 1$, and therefore $C_m \sim \max(P, M) = M \sim 2^{n/2}$ and $S = 1$, and $T \sim 2^{n/2}$ resulting in $C_s \sim 2^n$ which is the same as exhaustive search. Because in our model the cost of the machine is determined by $\max(P, M)$, it is cost-effective to use the same number of processors as words of memory. Using the architecture and the algorithm suggested in this section we can obtain point B that has $P \sim M \sim 2^{n/2}$, $C_m \sim \max(P, M) \sim 2^{n/2}$, $S = 1$, $T \sim 1$ and, therefore, $C_s \sim 2^{n/2}$ which is much better than the usual time–memory trade-off.

2) As noted earlier Schroeppel and Shamir's time–memory trade-off achieves a portion of the curve in Fig. 4, from $\alpha = 1$ to $\alpha = 1/2$ [10]. By applying parallelism to their method, we can get (C_m, C_s) points from $\alpha = 1/2$ to $\alpha = 1/3$ [9]. Because their method uses a smaller number of processors, it is more practical for $2^{2n/3} \le C_s \le 2^n$, but we have not been able to extend it to smaller values of C_s.

3) We have found other parallel algorithms that give portions of the curve $C_m C_s \sim N$ [9]. All of these algorithms had performances no better than the previous algorithm, indicating that the $(C_m - C_s)$ curve might be the best possible for the binary knapsack. Recently, however,

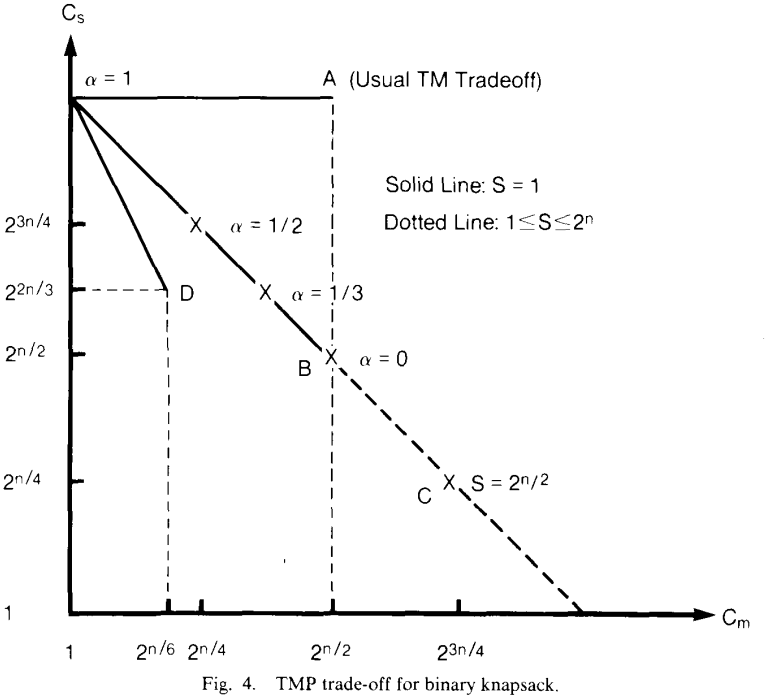

Fig. 4. TMP trade-off for binary knapsack.

Karnin [11] found an algorithm that has $C_m \sim 2^{n/6}$, $T \sim 2^{n/2}$, $S=1$, and hence $C_s \sim 2^{2n/3}$, resulting in $C_m C_s < 2^n$. This is plotted as point D. Karnin's algorithm can be generalized to give the best known result for the entire solid line between $(C_m = 1, C_s \sim 2^n)$ and $(C_m \sim 2^{n/6}, C_s \sim 2^{2n/3})$, but it has not been extended to values of C_s, lower than $2^{2n/3}$.

V. Time–Memory–Processor Trade-Off for Multiple Encryption

In [6] Diffie and Hellman analyzed the security of a simple double encryption scheme. For concreteness they considered a specific cryptosystem such as DES. Let P_0 be a 64-bit plaintext, C_0 a 64-bit ciphertext, and K a 56-bit key. The basic DES operation can be represented as

$$C_0 = E_K(P_0)$$

and simple double encryption is obtained as

$$C_0 = E_{K2}(E_{K1}(P_0)).$$

Diffie and Hellman used a "meet in the middle" attack (assuming a known plaintext–ciphertext pair) that breaks the foregoing cipher in about 2^{56} operations and 2^{56} words of memory. While they effected a reduced cost compared to exhaustive search, under the simplified analysis of this paper their method corresponds to a cost per solution of 2^{112} ($C_m \sim 2^{56}$, $S=1$, $T \sim 2^{56}$, and $C_s \sim 2^{112}$) which is the same as exhaustive search ($C_m = 1$, $S=1$, $T \sim 2^{112}$, and $C_s \sim 2^{112}$).

In our simplified model, C_m is determined by $\max(P, M)$, so it pays to consider an algorithm that uses the same number of processors and words of memory. In this section we shall develop an algorithm for the double encryption problem that, like the knapsack problem, has $S=1$ for half of the $(C_m - C_s)$ curve. Neglecting constant and logarithmic factors, this algorithm corresponds to a TMP trade-off with

$$C_m \sim 2^{(1-\alpha)n} \qquad S=1 \qquad T \sim 2^{2\alpha n} \qquad C_s \sim 2^{(1+\alpha)n}, \quad \text{for } 0 \le \alpha \le 1. \quad (5)$$

Note that, as before, the foregoing trade-off has $C_m C_s \sim 2^{2n} = N$. (Because there are two subkeys, each n-bits long, the total key is $2n$-bits long.) However, this is a better trade-off than the general TMP trade-off of Section III because it allows a lower simultaneity.

To get a cost per solution of less than $2^n = \sqrt{N}$, we must have $S > 1$. However, unlike the knapsack problem, the algorithm for this part does not allow a smooth increase in S, i.e., the simultaneity has a jump from 1 to $\sqrt{N}$. Therefore, the trade-off for $C_s < \sqrt{N}$ corresponds exactly to the general TMP of Section III, and in the following we only describe the algorithm for $C_s > \sqrt{N}$ and $S=1$.

The basic idea is to apply parallelism to the "meet in the middle" attack. We break K_1 and K_2 each into two parts of lengths αn and $(1-\alpha)n$, for $0 \le \alpha \le 1$. We fix the first α_n bits of K_1 and K_2 and simultaneously encrypt P_0 under the remaining $2^{(1-\alpha)n}$ values of K_1, decrypt C_0 under the remaining $2^{(1-\alpha)n}$ values of K_2, and use the sorting/switching network to check for a match. We need to do this $2^{2\alpha n}$ times to exhaust all possible combinations from the first parts of K_1 and K_2, resulting in (5). The details are given in the following algorithm where again $\boldsymbol{i}$ represents the binary vector obtained from the binary representation of the integer i.

1) Initialize:
   $i_1 = 0$ [first part K_1 of length αn bits]
   $i_2 = 0$ [first part K_2 of length αn bits]
   $M_1 = 2^{\alpha n}$
   $M_2 = 2^{(1-\alpha)n}$.
2) Fix the first parts of K_1 and K_2:
   IF $(i_1 < M_1)$, THEN GOTO step 4;
   ELSE GOTO step 3.
3) Increment i_2:
   $i_2 = i_2 + 1$
   $i_1 = 0$
   IF $(i_2 < M_1)$, THEN GOTO step 4;
   ELSE "no solution."
4) Encrypt P_0 and decrypt C_0 under all M_2 possible values of K_1 and K_2 (subject to the constraint that their first αn bits are i_1 and i_2, respectively).
   FOR $j = 0$ to $M_2 - 1$, DO
   $P_j = \langle E_{i_1 \# j}(P_0), j, \text{"encrypt"} \rangle$
   $P_{M_2 + j} = \langle E^{-1}_{i_2 \# j}(C_0), j, \text{"decrypt"} \rangle$.
5) Check to see if there is a match.
   a) Sort the processor values on the first field (using the sorting/switching network) and store them in the memory locations 1 through $2M_2$.
   b) Search the memory for adjacent items with the same value and of the form:
   $\langle \text{value}, j_1, \text{"encrypt"} \rangle$
   $\langle \text{value}, j_2, \text{"decrypt"} \rangle$
6) Output the solution or iterate the algorithm:
   IF (match exists), THEN:
   $\hat{\boldsymbol{K}}_1 = \boldsymbol{i}_1 \# \boldsymbol{j}_1$
   $\hat{\boldsymbol{K}}_2 = \boldsymbol{i}_2 \# \boldsymbol{j}_2$;
   GOTO step 7;
   IF (no match), THEN:
   $i_1 = i_1 + 1$
   GOTO step 2.
7) Check to see if $\hat{\boldsymbol{K}}_1$ and $\hat{\boldsymbol{K}}_2$ are the correct keys by encrypting an additional plaintext-ciphertext pair. If not, $\hat{\boldsymbol{K}}_1$ and $\hat{\boldsymbol{K}}_2$ represent a false alarm, in which case the algorithm is continued:
   $i_1 = i_1 + 1$
   GOTO step 2.

The algorithm uses $O(2^{(1-\alpha)n})$ processors and words of memory. The sorting/switching network requires $O(2^{(1-\alpha)n}\log^2 2^{(1-\alpha)n})$ elements [7]. Thus neglecting constant and logarithmic factors, the cost of the machine is $C_m \sim 2^{(1-\alpha)n}$. The simultaneity S is obviously one since only one problem is solved.

Step 4 of the algorithm takes unit time because it is done in parallel. Steps 5 and 6 take logarithmic time [7]. Therefore, neglecting constant and logarithmic factors, each iteration of the algorithm takes only unit time. Because there are $2^{2\alpha n}$ iterations, the total time T is $2^{2\alpha n}$. Therefore, the cost per solution $C_s \sim C_m T/S \sim 2^{(1-\alpha)n}2^{2\alpha n} = 2^{(1+\alpha)n}$. Note that for $\alpha = 0$ we have $C_m \sim 2^n = \sqrt{N}$, $S = 1$, $T \sim 1$, and $C_s \sim 2^n = \sqrt{N}$. This shows that under our simplified model a simple double encryption allows the same cost per solution as exhaustive search on a single encryption system. Of course, it requires a highly expensive machine since C_m is also $2^n = \sqrt{N}$, but this cost is offset by its very high speed.

Because $C_m C_s \sim 2^{2n} = N$ this TMP trade-off as shown in Fig. 5 has the same $(C_m - C_s)$ curve as the general TMP trade-off of Section III. However, the new algorithm is a better trade-off since it has $S = 1$ and hence a shorter time to find the solution.

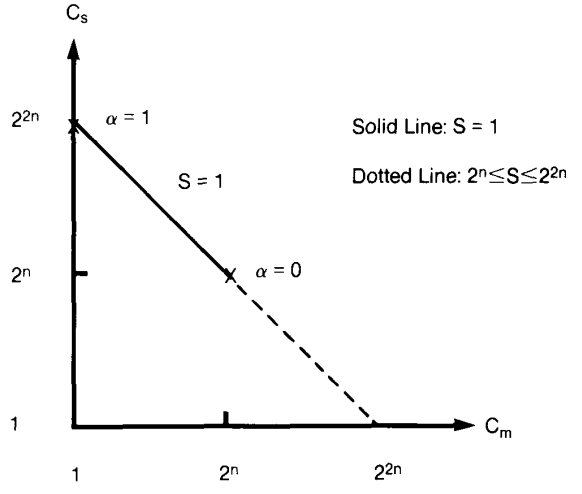

Fig. 5. TMP trade-off for double encryption.

Remarks: 1) The dotted portion of Fig. 5 corresponds to $S > 1$. Unlike the binary knapsack problem we were unable to find a smooth extension of the solid curve, i.e., the simultaneity S has a jump from 1 to $2^n = \sqrt{N}$ in the middle of the curve. Therefore, the $(C_m - C_s)$ curve for the dotted part has the same simultaneity as the one obtained from the general TMP trade-off of Section III.

2) The foregoing algorithm can be generalized to an m-level encryption, i.e., $C_0 = E_{Km}(\cdots E_{K2}(E_{K1}(P_0)))$ which under a known plaintext attack gives a trade-off between C_m and C_s such that $C_m C_s \sim 2^{mn} = N$, but as opposed to the trade-off of Section III it has $S = 1$ for $\sqrt{N} \le C_s \le N$. The details of this algorithm along with other trade-offs assuming a chosen text attack are given in [9].

VI. Discussion

Time–memory trade-offs have been considered as a cost-effective way of solving certain searching problems. However, we have shown that under our simplified model the usual time–memory trade-off has the same asymptotic cost per solution as exhaustive search. Even under an exact cost model, where the cost of a processor c_p is greater than the cost of a word of memory c_m, usual time–memory trade-offs can only cut the cost per solution by the ratio c_p/c_m.

Our time–memory–processor trade-off can be used when larger cost savings are desired, and when one has a large number of problems to solve. We have proposed an architecture consisting of a large number of processors sharing a large memory through a sorting/switching network. Under our simplified model, which neglects the difference between the cost of a processor and a word of memory, we can use an equal number of processors and words of memory. The processors are used to speed up the computation and/or work on many problems at the same time. Using this architecture we have developed a highly parallel and cost-effective algorithm for solving any searching problem. This algorithm allows a trade-off between the cost of the machine C_m and the cost per solution C_s, i.e., doubling the cost of the machine quadruples the solution rate and, therefore halves the cost per solution.

The additive structure of the binary knapsack allows an even better trade-off than the general TMP trade-off described earlier. While it has the same $(C_m - C_s)$ curve, it allows a lower simultaneity and hence a shorter time delay to find a solution, and using Karnin's approach, it is possible to achieve $C_m C_s < N$ for a portion of the curve.

We have also found a TMP trade-off for cryptanalyzing double encryption systems. This algorithm has $S = 1$ for half of its $(C_m - C_s)$ curve. The algorithm can be generalized to an m-level encryption and indicates that, in general, multiple encryption is less secure than a system with a single key of the same total size.

Throughout this paper we have neglected the difference between the cost of a processor and a word of memory (i.e., c_p versus c_m). This introduces a constant factor of c_p/c_m in the cost per solution which is about 10^4–10^6 with mainframe technology. However, with VLSI technology and special purpose processors this factor can be as small as 10–100 which is less important, especially when solving problems of large size.

A more fundamental problem is the cost of the central sorting/switching network. It is shown in [7] and [9] that such a network requires $O(N\log^2 N)$ elements, with $O(\log^2 N)$ stages. Therefore, the sorting/switching network increases the cost of the machine by a logarithmic factor; the delay through the network (or the number of the stages) is also a logarithmic factor, and hence can be neglected. However, as the problem size becomes ever larger, the dominant cost will be the wirings and the

propagation delays through these wires [9]. This problem requires further study.

Our TMP trade-off might be applied to other areas, such as the VLSI packing problem: given a large number of circuit elements, arrange them in as small an area as possible. It would be significant if the architecture and the algorithm given for the knapsack (one-dimensional exact fit problem) could be extended to VLSI packing (two-dimensional best fit problem).

An implication of our TMP trade-off is that one large machine is more cost-effective than two smaller machines: "big is better." This suggests computer networks with very large specialized machines for solving certain kinds of searching problems (e.g., cryptanalysis, knapsack, and VLSI design problem).

ACKNOWLEDGMENT

The authors are thankful to N. Pippenger of IBM and R. Winternitz of Stanford and the reviewers for many helpful suggestions.

REFERENCES

[1] R. C. Merkle and M. E. Hellman, "Hiding information and signatures in trapdoor knapsacks," *IEEE Trans. Inform. Theory.*, vol. IT-24, pp. 525–530, Sept. 1978.

[2] S. C. Pohlig and M. E. Hellman, "An improved algorithm for computing logarithms over GF(p) and its cryptographic significance," *IEEE Trans. Inform. Theory*, vol. IT-24, pp. 106–110, Jan. 1978.

[3] M. E. Hellman, "A cryptanalytic time-memory trade-off," *IEEE Trans. Inform. Theory*, vol. IT-26, pp. 401–406, July 1980.

[4] *Federal Register*, vol. 40, no. 149, Aug. 1, 1975.

[5] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Trans. Inform. Theory*, vol. IT-22, pp. 644–654, Nov. 1976.

[6] W. Diffie and M. E. Hellman, "Exhaustive cryptanalysis of the NBS data encryption standard," *Computer*, vol. 10, pp. 74–84, June 1977.

[7] K. Batcher, "Sorting networks and their applications," in *Proc. 1968 SJCC*, 1968, pp. 307–314.

[8] M. J. Marcus, "The theory of connecting networks and their complexity: A review," *Proc. IEEE*, vol. 65, no. 9, Sept. 1977.

[9] H. R. Amirazizi, Ph.D. dissertation, Electrical Engineering Dept., Stanford Univ., Stanford, CA, June 1986.

[10] R. Schroeppel and A. Shamir, "A $T = O(2^{n/2}), S = O(2^{n/4})$ algorithm for certain NP-complete problems," *SIAM J. Comput.*, vol. 10, pp. 456–464, Aug. 1981.

[11] E. D. Karnin, "A parallel algorithm for the knapsack problem," *IEEE Trans. Comput.*, vol. C-33, no. 5, pp. 404–408, May 1984.